Harnessing the Power of GPUs without Losing Abstractions in SAC and ARRAYOL: A Comparative Study

Jing Guo*, Wendell Rodrigues†, Jeyarajan Thiyagalingam‡,
Frédéric Guyomarch†,
Pierre Boulet†, and Sven-Bodo Scholz*
*Department of Computer Science
University of Hertfordshire
Hatfield, United Kingdom
Email: j.guo@herts.ac.uk
†LIFL - USTL - INRIA Lille Nord Europe, France
‡Oxford e-Research Centre, University of Oxford
Oxford, United Kingdom

***Abstract*—Over recent years, using Graphics Processing Units (GPUs) has become as an effective method for increasing the performance of many applications. However, these performance benefits from GPUs come at a price. Firstly extensive programming expertise and intimate knowledge of the underlying hardware are essential for gaining good speedups. Secondly, the expressibility of GPU-based programs are not powerful enough to retain the high-level abstractions of the solutions. Although the programming experience has been significantly improved by existing frameworks like CUDA and OPENCL, it is still a challenge to effectively utilise these devices while still retaining the programming abstractions.**

To this end, performing a source-to-source transformation, whereby a high-level language is mapped to CUDA or OPENCL, is an attractive option. In particular, it enables one to retain high-level abstractions and to harness the power of GPUs without any expertise on the GPGPU programming.

In this paper, we compare and analyse two such schemes. One of them is a transformation mechanism for mapping a image/signal processing domain-specific language, ARRAYOL, to OPENCL. The other one is a transformation route for mapping a high-level general purpose array processing language, Single Assignment C (SAC) to CUDA. Using a real-world image processing application as a running example, we demonstrate that albeit the fact of being general purpose, the array processing language be used to specify complex array access patterns generically. Performance of the generated CUDA code is comparable to the OPENCL code created from domain-specific language.

***Keywords*-ARRAYOL, SAC, GPGPU, CUDA, OPENCL**

I. INTRODUCTION

Over recent years, using Graphics Processing Units (GPUs) has become increasingly popular and in fact important for seeking performance benefits in computationally intensive parallel applications. The relative measure of performance/price and performance/power ratios between GPU-based architectures and CPU-based architectures further encourages the choice of GPUs. Typical modern GPUs contain hundreds of computational cores and have become one of the most commonly used many-core architectures. The architectural aspects of these GPUs are far more complex than mainstream multi-core CPUs and programming is often facilitated by programming models such as CUDA [5] and OPENCL [9]. At present, CUDA and OPENCL are the most widely used programming models for programming GPUs.

However, these performance gains are not without challenges: Firstly, the identification and exploitation of any parallelism in the application is the responsibility of programmers. Often, this requires intimate understanding of the hardware and extensive re-factoring work rather than simple program transformations. Secondly, the high-level abstractions of a problem can hardly be expressed in the CUDA or OPENCL programming model. Furthermore, subsequent manual optimisations distort any remaining abstractions in the application.

The OPENCL [9] programming model aims to address the portability issue and offers a route for separating the device logic from the application logic. However, the application logic cannot be entirely free from low-level device logic. Compiler directive-based approaches such as *hi*CUDA [8] or support from compilers, such as PGI [14], have enabled application developers to retain application logic in the source language such as C and/or Fortran. Essentially this approach eliminates low-level device-specific logic from the application but expects the application developers to hint the compiler. Although this is a significant improvement in the direction of offering a simple programming model, the developer is still required to be familiar with the hardware to provide hints.

There are several ways of addressing this abstraction vs performance issue in applications. One approach is to rely on source-to-source transformations [4], [6], whereby a high-level language is translated into OPENCL or CUDA source by an external compiler. This compiler-based approach enables the exploitation of GPUs while retaining

high-level abstractions of the applications. However, the ability to get good performance relies on the capability of the underlying compiler. The nature of the source language, whether it is a domain-specific or general-purpose language, partly determines the scope of the transformations that the underlying compiler can perform.

Another route is to provide a model-to-source transformation mechanism where the model is captured through a Model-Driven Environment (MDE) and then subsequent code generation is handled by templates. The front-end will capture and retain the abstractions, while the code-generation phase will help *partly* addressing the performance issues. The reason we say that it only partly addresses the performance issues is that the approach is different from compiler, and may not perform even simple optimising transformations.

In this paper we compare and analyse two such approaches. The first scheme is a MDE-based code generation mechanism where a image processing model is mapped to OPENCL. Underlying basic building blocks of these domain-specific operations are specified in a specification language, ARRAYOL. The second scheme maps a general purpose, functional array programming language, Single Assignment C(SAC), to CUDA. In this setup, the optimisation space is relatively unrestricted to any domain and therefore often conservative.

By using a simple image processing application, H.263 video compression, as a running example, we compare, analyse and share our experiences in resorting to these two approaches for harnessing the power of GPUs.

The rest of this paper is organised as follows: In Section II we provide a short background on both of programming languages. Section III discusses the image compression problem. Section IV discusses how the decoding is implemented in the ARRAYOL language, which is then followed by the description of the mapping process from ARRAYOL to OPENCL. This is then followed by the decoding process in SAC. The SAC to GPU mapping process is described in Section VII. These two schemes are then evaluated for their performance benefits and corresponding results are reported in Section VIII. We finally conclude the paper in Section IX.

II. SOURCE LANGUAGES

A. ARRAYOL

ARRAYOL [1] is a specification language for formalising multi-dimensional signal processing operations as array transformations. The specification is a two-fold process, known as GILR (Globally Irregular, Locally Regular). The overall processing transformation is described at the global level using a graph whose nodes exchange multidimensional arrays. The *local* operations performed within nodes are then expressed separately at the node level.

Since, ARRAYOL is only a specification language, no rules are specified for executing an application described with ARRAYOL, but a scheduling can be easily computed using this description. The following are the basic principles underlying the ARRAYOL language:

- ARRAYOL can be considered as a first order functional language, single-assignment language. The static single assignment formalism ensures that no data element is ever written twice although it can be read several times.
- In ARRAYOL, only the true data dependences are expressed in order to express the full parallelism of the application, defining the minimal order of the tasks. Thus any schedule respecting these dependences will lead to the same result and thus the language is deterministic.
- All the potential parallelism in the application has to be available in the specification, both task parallelism and data parallelism.
- Data accesses are done through sub-arrays, called patterns.
- The language is hierarchical to allow descriptions at different granularity levels and to handle the complexity of the applications. The data dependences expressed at a level (between arrays) are approximations of the precise dependences of the sub-levels (between patterns).
- The spatial and temporal dimensions are treated equally in the arrays. In particular, time is expanded as a dimension (or several) of the arrays. This is a consequence of single assignment.

The semantics of ARRAYOL is that of a first order functional language manipulating multidimensional arrays. Although it is not a data flow language, it can be projected on such a language.

B. Single Assignment C (SAC)

SAC, in contrast to ARRAYOL, is a full, standalone functional and data-parallel programming language. Most of its basic language constructs are identical to those of C, not only with respect to their syntax but also with respect to their semantics. Despite this rather imperative look and feel, a side-effect free semantics is enforced by the exclusion of a few features of C, most notably the notion of pointers. As a replacement, the language incorporates extensive support for compiler-managed multi-dimensional arrays.

The language features an elegant yet powerful construct, WITH-loop, for expressing data-parallel operations. The formal syntax of a WITH-loop is sketched in Figure 1. Note here that we look at a simplified form that suffices to explain the main aspects of this paper. A complete discussion can be found in [13].

A WITH-loop expression in SAC consists of one or more generator parts and an operation. The latter determines the overall behaviour of a WITH-loop. In the context of this paper we only look at operations that have the forms `genarray(` $Expr$ `)` and `modarray(` $Identifier$ `)`. They define the creation and modification of an array where

$$
\begin{array}{lcl}
Expr & \Rightarrow & \ldots \\
 & | & \textbf{with}\ \{\ \lceil\, Generator : Expr;\ \rceil^{+}\ \} : \\
 & & \quad Operation \\
Generator & \Rightarrow & (\ Expr <= Id < Expr \lceil\, Filter\ \rceil\ ) \\
Filter & \Rightarrow & \textbf{step}\ Expr\ \lceil\ \textbf{width}\ Expr\ \rceil \\
Operation & \Rightarrow & \textbf{genarray}\ (\ Expr\ ) \\
 & | & \textbf{modarray}\ (\ Identifier\ )
\end{array}
$$

Figure 1. Syntax of WITH-loop in SAC

$Expr$ is the new array shape and $Identifier$ represents the array to be modified. The elements of the result array are defined by the generators, which specify mappings from indices to values. Each generator specifies an index range and an expression that is to be evaluated for each index within that range.

Various application studies have demonstrated that the compiler generated codes can achieve: (i) competitive sequential runtimes comparable to those of hand-written C and FORTRAN codes, and (ii) almost linear speedups from auto-parallelisation for shared memory systems [3]. More detailed introductions into SAC can be found in [13].

III. Case Study: H.263 Video Compression

The case study concerned in this paper deals with specific aspect of H.263-based video compression standard, scaling. The scaling during video-compression is considerably important for previews or for streaming for small form factor devices, such as mobile phones. The application consists of a classical downscaler, which transforms a video signal, which, for instance, is expressed in Common Intermediate Format (CIF), into a smaller size video. In this situation, the downscaler can be composed of two components: a horizontal filter that reduces the number of pixels from 352-lines to 132-lines and a vertical filter that reduces the number of pixels from 288-lines to 128-lines by interpolating packets of 8 pixels both row- and column-wise.

In a typical case of handling a 25-frames-per-second video signal lasting for 80-seconds, the downscaler may process up to 2000 frames in CIF format, with each input frame being represented by a two-dimensional array of size 352×288 and should emit 2000 output frames of size 132×128. Since each video pixels is encoded in 24-bit RGB colour model, the frame generation process is repeated for each frame and for each pixel of different colour space along two different directions. The final frame is produced by using these outputs from different colour space. Depending on the composing function, a broad-range of output colours are possible for each pixel and thus for each frame. The Figure 2 illustrates this basic operation for a given frame in high-definition format.

As can be observed, the operations concerned with the scaling is highly parallel and repetitive. The interpolation is repeated for each frame, each pixel and for each colour channel.

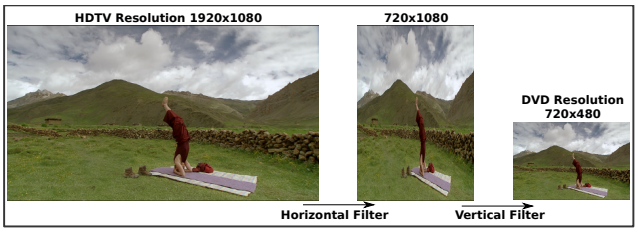

Figure 2. Horizontal and vertical filter processes

IV. Image Compression in ArrayOL

As mentioned in Section I and in Section II, ARRAYOL is a specification language, which underpins the exact operational and coordination aspects of the functional blocks captured by the MDE. In particular, the scaling operation, when performed in parallel, boils down to specifying scatter/gather operations, specifying the ARRAYOL patterns, requirements which are well captured by the ARRAYOL language. The distributed data-flow aspect of the down-scaler is illustrated in the figure 3.

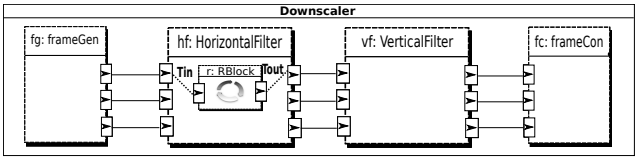

Figure 3. Downscaler overview

One of the fundamental aspects of the ARRAYOL language is a *tiler*, a special connector which binds input and output ports of different operational blocks. In other words, tilers represent data flow between components. For example, in Figure 3, each iteration of the repetitive task instance *r* of the *RBlock* task definition has its ports connected to external ports by special connectors called *tilers*.

The *tiler* expresses how multidimensional arrays are tiled by patterns. When applied to a connector, a *tiler* connects an external port with a port of an internal part. The shape of the external ports defines the shape of input/output arrays of a task. The port shape of the internal part defines the pattern shape and the shape of the part itself defines the repetition space. A tiler uses three main information to define the tiling operation:

- origin vector $\boldsymbol{o}$, which specifies the origin of the reference tile in the array;
- fitting matrix $\boldsymbol{F}$, which specifies how the patterns are filled with array elements;
- paving matrix $\boldsymbol{P}$, which specifies how an array is covered by pattern elements.

Thus, we can summarise the *tiler* operations as follows:

- $\forall i, o \leq i < s_{pattern}, e_i = o + F.i \bmod s_{array}$ enumerates all the elements of a pattern (e_i) for each i in the pattern shape.
- $\forall r, o \leq r < s_{array}, ref_r = o + P.r \bmod s_{array}$ gives all the reference elements of the patterns, s_{array} being the shape of the repetition domain.

V. Compiling ArrayOL to OpenCL

A. MDE and MARTE

Model Driven Engineering (MDE) [10] aims to raise the level of abstraction in program specification and increase automation in program development. The UML profile for MARTE [11] extends the possibilities for modelling of application and architecture and their relations. MARTE consists in defining foundations for model-based description of real time and embedded systems. The MARTE profile provides ArrayOL support in the *repetitive structure modelling* (RSM) package.

B. Model Transformation Chain

In MDE, a model transformation is a compilation process which transforms a source model into a target model. This allows for adding, modifying, transforming model elements in order to achieve a final model closer to the real program application. For instance, transformed models have explicit information about variables and task scheduling. In [10] there is an overview about the tools used in *model-to-model* and *model-to-text* process.

C. OpenCL Code Generation

The approach that allows us generating OpenCL code is part of Gaspard2 [4] project. In design time, Gaspard2 uses MARTE [11] in order to define a semantics to application project. Then, using transformation chains, it allows us to generate code for a few target languages and platforms such as OpenMP, SystemC, VHDL, Pthread and OpenCL. One of the main advantages of MARTE is that it clearly distinguishes the hardware components from the software components. This is done via stereotypes provided in part by the *Detailed Resource Modelling* (DRM) package, in particular the *HwResource* and *SwResource* stereotypes. For hybrid (CPU + Compute Device) conception, this separation is of prime importance as it is usual to create those two parts of the system simultaneously by different teams. Moreover, this separation provides a flexible way to independently change the software part or the hardware part. For instance, this allows for testing the software on different kind of hardware architecture, or to reuse an architecture (with a few or no changes) for different applications.

VI. Image Compression in SaC

Since tiler is the most important abstraction in ArrayOL to express sophisticated data access patterns, we aim to implement the same mechanism in SaC using WITH-loops and maintain a high level abstraction. Both the horizontal and vertical filters in the downscaler example use two tilers, one for gathering a pattern of data from an input frame and the other for scattering the data to an output frame after copmutation. We refer to these two tilers as *Input Tiler* and *Output Tiler*. In the following discussion, we will only look at horizontal filter while the same techniques can be applied to the vertical filter equally. In the context of SaC, the compression process can be performed in three steps:

- **Step 1**: The input tiler generates an intermediate array containing data gathered from the original frame. Shape of this intermediate array is a concatenation of the repetition space shape and the input pattern shape.
- **Step 2**: A WITH-loop performs the compression task with the intermediate array in a data-parallel fashion. This generates another intermediate array whose shape is a concatenation of the repetition space shape and the output pattern shape.
- **Step 3**: The output tiler scatters data from the second intermediate array to the output frame.

```
int[*] input_tiler(int[*] in_frame, int[.] in_pattern,
                   int[.] repetition, int[.] origin,
                   int[.,.] fitting, int[.,.] paving)
{
  output = with {
    (. <= rep <= .) {
     tile = with {
       (. <= pat <= .) {
        off = origin +
              MV( CAT( paving, fitting) , rep++pat);
        iv = off % shape(in_frame);
        elem = in_frame[iv];
       } : elem;
     } : genarray( in_pattern, 0);
    } : tile;
  } : genarray( repetition);
  return( output);
}
```

Figure 4. Input tiler function in SaC

Both the input and output tilers need to be specified generically so that they can be reused other access patterns. Figure 4 shows a function in SaC implementing the generic input tiler. The three main tiler components, origin vector, fitting matrix and paving matrix, are passed as arrays with generic shapes to the function. The function contains a WITH-loop nest to generate an intermediate array `output` from the input `in_frame`. The outer WITH-loop iterates the entire repetition space while the inner WITH-loop generates a single tile during each iteration. To gather data for a tile, an index vector `iv` for accessing the input frame is calculated from the origin, paving, fitting and pattern arrays. Calculation of `iv` is based on the formulae described in Section IV. Note that `MV` and `CAT` are functions performing matrix-vector multiplication and array concatenation respectively.

The intermediate array is then passed to the task function where a WITH-loop performs the actual frame compression (See Figure 5). During each iteration of the repetition space, an output tile is constructed from the corresponding input tile. This function creates a second intermediate array, `output`, whose shape is a concatenation of the repetition space shape `repetition` and the output pattern shape `out_pattern`.

Finally, the output tiler scatters the data of the newly compressed frame back to an output frame `out_frame`. Figure 6 shows a generic output tiler function in SaC. It

```
int[*] task(int[*] input, int[.] out_pattern,
            int[.] repetition)
{
  output = with {
    (. <= rep <= .) {
      tile = genarray(out_pattern, 0);
      tmp0 = input[rep][0] + input[rep][1] +
             input[rep][2] + input[rep][3] +
             input[rep][4] + input[rep][5];
      tile[0] = tmp0 / 6 - tmp0 % 6;
      tmp1 = input[rep][2] + input[rep][3] +
             input[rep][4] + input[rep][5] +
             input[rep][6] + input[rep][7];
      tile[1] = tmp1 / 6 - tmp1 % 6;
      tmp2 = input[rep][5] + input[rep][6] +
             input[rep][7] + input[rep][8] +
             input[rep][9] + input[rep][10];
      tile[2] = tmp2 / 6 - tmp2 % 6;
    } : tile;
  } : genarray( repetition);
  return( output);
}
```

Figure 5. Task function in SAC

```
int[*] generic_output_tiler(int[*] out_frame,
  int[*] input, int[.] out_pattern, int[.] repetition,
  int[.] origin, int[.,.] fitting, int[.,.] paving)
{
  for( i=0; i< repetition[[0]]; i++) {
    for( j=0; j< repetition[[1]]; j++) {
      for( k=0; k< out_pattern[[0]]; k++) {
        off = org + MV( CAT(paving, fitting), [i,j,k]);
        iv = off % shape( out_frame);
        out_frame[iv] = input[[i,j,k]];
      }
    }
  }
  return( out_frame);
}
```

Figure 6. Generic output tiler in SAC

contains a *for*-loop nest iterating through each element of the intermediate array `input`. Similar to the input tiler, each element is scattered to a specific index position in the output frame determined by the formulae described in Section IV.

```
int[*] nongeneric_output_tiler(int[*] output,
                               int[*] input
{
  output = with {
    ([0,0]<=[i,j]<=. step [1,3]):input[[i,j/3,0]];
    ([0,1]<=[i,j]<=. step [1,3]):input[[i,j/3,1]];
    ([0,2]<=[i,j]<=. step [1,3]):input[[i,j/3,2]];
  } : modarray( output);
  return( output);
}
```

Figure 7. Non-generic output tiler in SAC

This generic implementation can be used by applications with different tiler specifications. However, as we will see in the next section, nested *for*-loops in SAC can potentially hinder effective optimizations of the compiler. To be able to evaluate the performance impact of different programming abstractions, we implemented a non-generic output tiler shown in Figure 7. In this implementation, we assume prior knowledge of output tile size (i.e. 3 in the case of horizontal filter). One WITH-loop generator is created for accessing all elements at the same index position within a tile. Each generator iterates along the data scattering dimension in step equal to the tile size (i.e. `step [1,3]`) and the accessing index of that dimension is the quotient of the corresponding WITH-loop index and the tile size (e.g. `j/3`).

VII. Compiling SAC to CUDA

As discussed in the previous section, the image compression can be logically divided into three distinct steps, each implemented by an SAC function. This abstract and modularised approach encourages code reuse at the expense of potentially compromising program performance. As we have seen, two intermediate arrays are created: one from the input tiler and the other from the task function. If they are actually allocated in memory, it will not only increase the total memory footprint but also incur expensive data copy and hinder effective data reuse.

```
int[1080, 1920] in_frame;
int[1080, 720] output;

output = with {
  ( [0,0] <= iv < [1080,1] step[1,3] width[1,1] {
    res1 = ...in_frame[...]...;
  } : res1;
  ( [0,1] <= iv < [1080,2] step[1,3] width[1,1] {
    res2 = ...in_frame[...]...;
  } : res2;
  ( [0,3] <= iv < [1080,720] step[1,3] width[1,1] {
    res3 = ...in_frame[...]...;
  } : res3;
  ( [0,4] <= iv < [1080,720] step[1,3] width[1,1] {
    res4 = ...in_frame[...]...;
  } : res4;
  ( [0,2] <= iv < [1080,1] step[1,3] width[1,1] {
    res5 = ...in_frame[...]...;
  } : res5;
} : genarray( [1080, 720]);
```

Figure 8. Code after WITH-loop folding

To minimize the performance overheads of using large intermediate arrays, the SAC compiler performs one crucial optimization - WITH-loop Folding (*WLF*) [12]. This optimization identifies consecutive WITH-loops with `Use-Def` relationship and fuses them aggressively. This renders allocation of intermediate arrays in memory unnecessary and more importantly, avoids expensive data copy and enables better data reuse. In the case of non-generic output tiler, we have three consecutive WITH-loops, each consuming the result of a previous one. This triggers *WLF* and eventually generates a single WITH-loop similar to the one shown in Figure 8 (we assume the frame size is 1080×1920 and omit the actual compression code inside each generator for clarity). On the other hand, *WLF* fails in the case of generic output tiler as it does not attempt to fuse program constructs other than WITH-loops (the output tiler is specified as a *for*-loop nest). Here, we essentially trade abstraction for performance and the these two approaches will be evaluated in Section VIII.

After performing all high level optimizations in SAC, the intermediate program is passed to the CUDA backend for generating actual GPU code. Instead of performing whole-

program transformation, we focus on translating individual data-parallel WITH-loops to equivalent CUDA kernel functions. Here we briefly outline the three main steps of this transformation process. A more thorough discussion of the formal compilation schemes can be found in [7]:

- Identifying WITH-loops eligible to be executed on GPUs (i.e. CUDA-WITH-loops). Inherent limitations of the CUDA architecture and the programming model, such a lack of support for stack and nested thread creation, render certain WITH-loops un-parallelisable. The CUDA backend therefore only parallelises the outermost WITH-loops containing no function invocations. This both satisfies the constrains and increases the granularity of parallelism to better amortize various GPU overheads (e.g. device initialisation and kernel launch).
- Inserting data transfer instructions for CUDA-WITH-loops. We introduce two dedicated instructions, `host2device` and `device2host` , to transfer data between host and device memory for both free variables and results of CUDA-WITH-loops. The orginal variables are then replaced by their device counterparts.
- Creating CUDA kernel functions from CUDA-WITH-loop. We outline each WITH-loop generator as a kernel function and replace it by the corresponding invocation. Kernel configurations are derived from the generator bounds. Code of each generator becomes the function body.

VIII. Evaluation

The main aim of our evaluation is to quantify the performance impacts of different programming models on modern heterogeneous architectures. For this purpose, we compile and execute the downscaler application operating on 1080×1920 image frames. The test system contains an Nvidia Fermi GTX480 GPU. The device has 15 streaming multiprocessors. Each multiprocessor has 32 streaming processors clocked at 1.4 GHz. The total amount of device memory is 1.5 GB. The CPU is an Intel 2.8 GHz i7-930 quad core processor with 8 MB L2 cache. The GPU is connected to the CPU through a PCIe x16 Gen2 bus. We use CUDA version 3.1 and enable *-O3* option for all compilations.

A. Performance Evaluation of the SAC Implementations

To compare the performance of the generic and non-generic downscaler implementations in SAC, we measure the runtimes of both horizontal and vertical filters, each executed for 300 iterations. We also compare the performance of both the sequential and CUDA code generated by the SAC compiler (denoted as SAC-Seq and SAC-CUDA respectively). The execution times of different implementations are shown in Figure 9. The first thing we can observe is that the CUDA code performs significantly better than its sequential counterpart. This is because each output frame

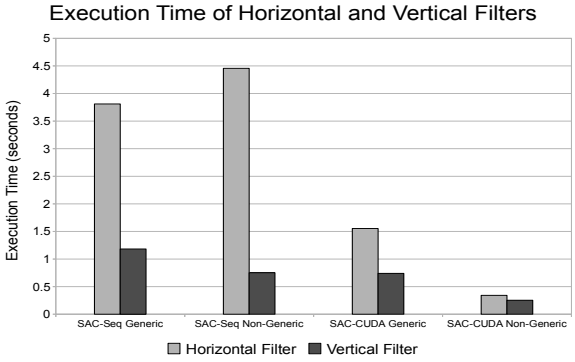

Figure 9. Filter execution times of different SAC implementations (300 iterations)

element can be computed independently, therefore providing abundant fine-grain parallelism for GPU's massive parallel architecture to exploit. Another interesting finding is that, while execution times of sequential code do not vary significantly between generic and non-generic implementations, the non-generic filters execute 4.5× (horizontal) and 3× (vertical) faster than the generic versions on GPU. As we discussed in Section VII, the generic output tiler is specified as a *for*-loop nest. Since the SAC compiler does not attempt to parallelise loops apart from WITH-loops, the *for*-loop nest is executed on the host. Since both the input tiler and task function are executed on the GPU and produce intermediate results in the GPU memory, the intermediate result has to be transferred back to the host memory before the output tiler can access it. This device-to-host transfer time significantly increases the total runtime of the filters. On contrary, the input tiler, task function and output tiler are fused into one single WITH-loop by the *WLF* optimization in the non-generic implementation. Therefore, it is executed on the GPU completely without any intermediate data transfers, improving performance dramatically.

B. Performance Evaluation of the GASPARD2 Implementation

The downscaler application in GASPARD2 is modeled in Papyrus [2] in the Eclipse environment using four macro tasks:

1) *FrameGenerator*, which is an elementary task linked to an IP (intellectual property) that reads frames from a video file or camera using the OpenCV library;
2) *HorizontalFilter*, which is hierarchically composed by three elementary tasks that become kernels in the GPU environment and it is responsible for horizontal scaling;
3) *VerticalFilter*, which is equivalent to *HorizontalFilter* for vertical scaling;
4) *FrameConstructor*, which is an elementary task linked to an IP that writes frames out to a file or display device using the OpenCV library.

The model of the *HorizontalFilter*, with some additional information about tilers and input and output data, is illustrated in the figure 10. This component is composed of three similar tasks that deals with *RGB* components of the video frame. For instance, the *bhf* (horizontal filter for *B* component) has a shape size of 1080x240 and this is the repetition space for the tiler specifications.

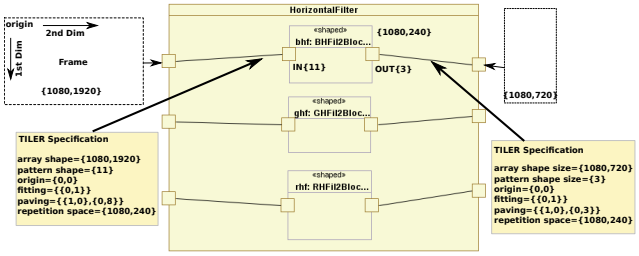

Figure 10. Downscaler tiler specification

Since we have the models defined in design time, a single step triggers the transformations and generates code for the chosen target. We use the downscaler model (UML file) in the Eclipse project tree navigator, then we execute the OPENCL chain. This produces source files (.cpp, .cl) and a makefile. The Figure 11 is part of a resulting kernel source code file. This part is related to the *tiler* creation and it is responsible to transfer the pattern elements from the global memory to the work-item private memory (latest lines).

```
//--- Tiler TFunc_in_bhf_bhf_KRNPAR::in_bhf_bhf ---
{ //start block
  uint tlIter[2];
  uint tl[1];
  uint ref[2];
  uint index[2];
  //get indexes based on work-item Global ID
  tlIter[0]=iGID%1080;
  tlIter[1]=abs(iGID/1080);
  //reference point based on Paving matrix
  ref[0] = 0 + 1*tlIter[0] + 0*tlIter[1];
  ref[1] = 0 + 0*tlIter[0] + 8*tlIter[1];
  //pattern filling based on Fitting matrix
  for(tl[0]=0; tl[0] < 11; tl[0]++) {
    index[0]= (ref[0] + 0*tl[0])%1080;
    index[1]= (ref[1] + 1*tl[0])%1920;
    in_bhf_bhf[tl[0] * 1] =
      in_bhf_bhf_KRNPAR[index[0] * 1920 +index[1] * 1];
  } //end for
} // end block
```

Figure 11. Generated code of a tiler

The running results for GASPARD2 version are showed in the Table I. More than half of the time is dedicated to data transfers, mainly in the beginning when the host needs to transfer an entire frame. We have three kernels to do the horizontal filter and three to do the vertical filter as well. The horizontal filter kernels take more time to execute due to their larger grid size. This application takes 2.86s to process 300 frames. This is suitable for real time playing of a 25 fps high-definition video that takes 12s to finish.

Operation	#calls	GPU time(μsec)	GPU time (%)
H. Filter (3 kernels)	300	844185	29.51
V. Filter (3 kernels)	300	424223	14.83
memcpyHtoDasync	900	1391670	48.74
memcpyDtoHasync	900	197057	6.89
Total	-	**2.86sec**	**100.00**

Table I
KERNEL EXECUTION AND DATA TRANSFER TIMES OF GASPARD2 IMPLEMENTATION

Operation	#calls	GPU time(μsec)	GPU time (%)
H. Filter (5 kernels)	300	1015137	29.60
V. Filter (7 kernels)	300	762270	22.22
memcpyHtoDasync	900	1454400	42.40
memcpyDtoHasync	900	198000	5.77
Total	-	**3.43sec**	**100.00**

Table II
KERNEL EXECUTION AND DATA TRANSFER TIMES OF SAC IMPLEMENTATION

C. Performance Comparison of SAC and GASPARD2

Table II shows a detailed breakdown of kernel execution time and data transfer time the non-generic SAC implementation takes to process 300 frames. Similar to the GASPARD2 implementation, data transfers represent approximately 50% of the total execution time. The reason is that both approaches transfer the same amount of frame data to the device memory before compression starts and back to the host memory afterwards for displaying. Figure 12 shows runtime comparison between these two approaches. As we can see, horizontal and vertical filters in GASPARD2 perform slightly better than SAC. Upon further investigation, we discover that each filter in GASPARD2 is specified as a single OPENCL kernel. As discussed in Section VII, the final fused WITH-loop for horizontal filter after applying *WLF* has 5 generators (the vertical filter has 7 generators). Since the CUDA backend creates one kernel for each generator, this means 5 kernels need to be launched during runtime. Such large number of kernel invocations is inefficient in two aspects and causes slowdowns of the SAC implementation:

- Each kernel launch incurs context overheads. The more kernels a program executes, the higher this cost will be.
- Data in certain memory of the GPU is not persistent

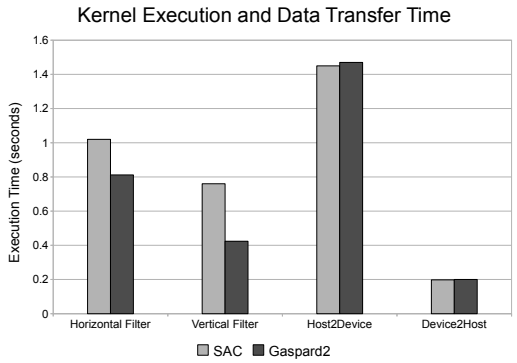

Figure 12. Comparison of the operation times

across different kernels, such as the on-chip L1 cache. Therefore, separating computations of the same data array into different kernels hinders effective data reuse.

IX. Conclusions

In this paper, using the H.263 video compression as a running example, we investigated two different approaches for retaining high-level abstractions in applications, while targeting modern GPUs for performance. One of our approaches is to rely on high-level, generic source-to-source transformation using a compiler. The other approach is to rely on a model-to-source code generation technique based on functional building blocks and a model-driven design environment. The former is a widely applicable technique crossing domain boundaries while the latter is tied to a specific domain, and in our case signal/image processing. Based on our programming experience and experimental evaluation, we make the following observations:

- Both techniques enable high-level program specification, including the expression of data parallelism. The ArrayOL language provides functional specification capturing the parallelism, which in turn can be leveraged from a front-end. The SaC programming language provides abstractions at a higher-level, with a self-contained full expression of the problem.
- Both approaches can lead users to a GPU-specific solution without detailed knowledge of the GPGPU programming. The ArrayOL-based approach leads to an OpenCL-based solution, while the SaC-based approach provides a CUDA-based solution.
- Despite the differences in the high-level program abstractions and in the final GPU-specific targets, performance benefits of both approaches are comparable, varying within 85% of the best runtimes.
- Upon further investigation, the ArrayOL-based approach leads to a solution with considerably fewer number of kernels than the SaC-based solution. This is mainly due to *WLF* in SaC. As a result, the ArrayOL-based solution achieves relatively better performance.

In our case, we were able to use a general purpose array programming language, SaC for successfully modelling a complex domain-specific problem expressed in a specification language, ArrayOL. Although compiler-driven optimisations often lead to benefits, we witnessed that in the context of GPGPU programming, they can equally add overheads such that simple model-to-source code generation technique can yield better results. Finally and most importantly, both the approaches can lead to significant performance benefits, as much as $11\times$ speedups on GPUs compared to sequential counterparts — all without losing the abstractions.

References

[1] A. Demeure, A. Lafage, E. Boutillon, D. Rozzonelli, J-C. Dufourd, J-L. Marro. Array-OL: Proposition dún Formalisme Tableau pour le Traitement de Signal Multi-Dimensionnel. In *Gretsi*, Juan-Les-Pins, France, September 1995.

[2] CEA. Papyrus - Open Source Tool for Graphical UML2 Modeling, Last accessed December 15, 2010. http://www.papyrusuml.org.

[3] R. Daniel, J. Carl, K. Alexei, S. Sven-Bodo, and V. S. Alexander. Numerical Simulations of Unsteady Shock Wave Interactions Using SaC and Fortran-90. In *Lecture Notes in Computer Science*, volume 5698, pages 445–456. Springer-Verlag, 2009.

[4] DaRT Team LIFL/INRIA, Lille, France. Graphical array specification for parallel and distributed computing (Gaspard2), Last accessed December 15, 2010. https://gforge.inria.fr/projects/gaspard2/.

[5] David B. Kirk, Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[6] Fred V. Lionetti and Andrew D. McCulloch and Scott B. Baden. Source-to-Source Optimization of CUDA C for GPU Accelerated Cardiac Cell Modeling. In *Euro-Par (1)*, volume 6271 of *Lecture Notes in Computer Science*, pages 38–49. Springer-Verlag, 2010.

[7] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Towards Compiling SaC to CUDA. In *Trends in Functional Programming*, volume 10, pages 33–48, Bristol, UK, 2010. Intellect.

[8] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: A High-Level Directive-Based Language for GPU Programming. In *GPGPU-2: Proceedings of 2nd Workshop on GPGPUs*, pages 52–61, New York, USA, 2009. ACM.

[9] Khronos Group. OpenCL 1.1, Last accessed November 22, 2010. http://www.khronos.org/opencl/.

[10] D. Lugato, J-M Bruel, and I. Ober. *Modeling, Simulation and Optimization - Focus on Applications*, chapter 2, pages 19–30. In-Tech, March 2010.

[11] OMG. Modeling and Analysis of Real-time and Embedded systems (MARTE), Last accessed December 15, 2010. http://www.omgmarte.org.

[12] Sven-Bodo Scholz. With-Loop-Folding in SAC – Condensing Consecutive Array Operations. In *In Proceedings of the 9th International Workshop on Implementation of Functional Languages*, Scotland, 1998. Springer-Verlag.

[13] Sven-Bodo Scholz. Single Assignment C – Efficient Support for High-level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[14] Wolfe, Michael. Implementing the PGI Accelerator model. In *GPGPU '10: Proceedings of the 3rd Workshop on GPGPUs*, pages 43–50, New York, USA, 2010. ACM.